

Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE



Jan Klíma

Systém pro vektorizaci rastrových obrázků

System for Image Vectorization

Katedra softwarového inženýrství

Vedoucí bakalářské práce: Doc. RNDr. Tomáš Skopal, PhD.

Studijní program: Informatika, obecná informatika

2007

I would like to thank my supervisor Tomáš Skopal for all the time he spent giving me invaluable feedback and support ideas.

I hereby certify that I wrote the thesis myself, using only the referenced sources. I agree with lending the thesis.

Prague, August 7, 2007

Jan Klíma

Contents

1	Introduction	6
2	IVP Framework	8
2.1	Component Layer	9
2.1.1	Primitive Types & Metadata	9
2.1.2	Interfaces	10
2.1.3	Components	11
2.2	Network Layer	13
2.2.1	Component Manager	13
2.2.2	Component Adapter	13
2.2.3	Component Network	14
2.3	XML Configuration	14
2.4	Application Level	15
2.5	Problems Related To Framework Design	16
2.5.1	Component Parametrization	16
2.5.2	Thumbnail Viewing	16
2.5.3	Line Drawing Representation & Handling	17
2.6	Framework Discussion	19
2.6.1	IVP Framework vs. Specialized Applications	19
2.6.2	Interfaces	19
2.6.3	Multiple Object Processing	20
3	Component Catalogue	21
3.1	Image Processing Components	21
3.2	Edge Detection Components	22
3.3	Segmentation Components	24
3.4	Binary Image Processing Components	25
3.5	Vector Processing Components	28
3.6	Line Drawing Output Components	32
4	Domain Scenarios	34
4.1	Map	34
4.2	Black And White Drawing	34
4.3	Logo	36
4.4	High Contrast Object	37
4.5	Complex Object	37
5	User Manual	40
5.1	Requirements	40
5.2	Installation	40
5.3	Uninstallation	40
5.4	Program execution	41
5.4.1	Console Application	41
5.4.2	Graphic User Interface Application	41
5.5	Graphic User Interface	41
5.5.1	Menu	41

5.5.2	Designer	43
5.5.3	Run On Change	43
5.5.4	Component Manager Panel	43
5.5.5	Property Page Panel	44
5.5.6	Thumbnails Panel	44
6	Comparison With Similar Programs	45
6.1	WinTopo 3.2 Professional	45
6.2	Vextractor 3.80	46
6.3	JaGrLib	47
7	Conclusion & Future Work	49
7.1	Framework-Related Improvements	49
7.2	GUI-Related Improvements	49
	Appendices	51
A	XML document example	51

Název práce: Systém pro vektorizaci rastrových obrázků

Autor: Jan Klíma

Katedra: Katedra softwarového inženýrství

Vedoucí bakalářské práce: Doc. RNDr. Tomáš Skopal, Ph.D.

E-mail vedoucího: Tomas.Skopal@mff.cuni.cz

Abstrakt: Extrakce tvarů je rozsáhlý a stále otevřený problém. Přestože je k dispozici celá řada existujících metod a algoritmů, není jasné, které z těchto metod použít a jak je zkombinovat ve snaze vytvořit robustní extrakční postup. Předmětem práce je komponentový systém pro vektorizaci rastrových obrázků navržený tak, aby bylo možné snadno konfigurovat různé průtoky dat sítí komponent. Na popis systému navazuje katalog komponent nabízející základní algoritmy pro zpracování obrázků, vektorizaci a úpravy vzniklého vektorového výstupu. Stručně jsou též nabídnuty možné scénáře zapojení a konfigurace komponent. Součástí práce je srovnání systému s jinými aplikacemi nabízejícími podobnou funkcionalitu.

Klíčová slova: vektorizace, extrakce tvarů, framework

Title: System for Image Vectorization

Author: Jan Klíma

Department: Department of Software Engineering

Supervisor: Doc. RNDr. Tomáš Skopal, Ph.D.

Supervisor's e-mail address: Tomas.Skopal@mff.cuni.cz

Abstract: The task of shape extraction is very complex and hasn't yet been satisfactorily solved. Although there exist lots of algorithms for shape extraction, the mere knowledge of such techniques doesn't give us an answer what methods should be used and how to combine them in order to obtain robust extraction method. This thesis is devoted to creating a component system for image vectorization, which is designed to allow easy configuration of shape extraction data-flows. The system's description is followed by a component catalogue that offers basic algorithms for image processing, vectorization and vector output handling. Component configurations as well as typical network scenarios are covered briefly. The proposed system is then compared to similar applications.

Keywords: vectorization, shape extraction, framework

1 Introduction

Shapes (object boundaries) present in an image can be regarded as medium-level features that usually carry important semantic information, shape recognition is also a crucial element of human sight. The ability to reliably extract shapes would represent an essential step in many areas, including computer vision, search in image databases, etc.

Shape Extraction

The most frequent shape extraction technique is to vectorize the contiguous lines or areas within the image into a set of vectors (polylines, polygons, arcs, bezier curves, ...). Prior to this step, the input raster needs to be preprocessed by various image processing algorithms such as smoothing, edge detection, thinning, etc. Vectorization step follows subsequently (typically, binary image is traced and vectors are extracted). Unfortunately, we have not finished our work yet - the obtained vector output contains lots of undesired vectors (as a result of noise, poor image quality, ...); removal of such artifacts needs to be tackled.

Shapes & Search in Image Databases

With the rapidly growing volumes of image data, similarity search in image databases is becoming even more important. Text-based image retrieval systems become useless as the requirements on manual annotation far exceed human possibilities. Metadata-based systems have similar problems, because they usually require some additional explicit information to describe images and this information may be unavailable. Generally, we need to consider the real content of each particular image to obtain good retrieval results.

In image databases, the task of content-based similarity search is extremely difficult due to completely unrestricted origination of contained raster images. The most general techniques of feature extraction are based on processing of low-level features, such as color histograms, color moments, etc. Such methods however consider only local or global relationships between the pixels and don't capture high-level information. In real world applications, there exist high-level feature extraction systems, but these are usually restricted to domain-specific databases (e.g. databases of biometric features); such systems cannot be used to manage heterogenous image collections.

When extracting shapes for the sake of similarity measuring, we are not at all interested in a comprehensive vector representation, instead we would like to capture the basic contained shape features and create a simple but powerful shape descriptor. The desired shape descriptor must be small enough (at most few kilobytes), not only to perform real-time similarity measuring but because otherwise we end up with huge clusters of meaningless vector data.

Thesis Contribution

As mentioned before, the process of shape extraction is a complex one, with almost no general recommendations about individual extraction steps. In this thesis,

a component-based framework is introduced that allows management of various image and vector processing algorithms. Shape extraction scenarios are created by connecting components into complex networks. A catalogue of basic components is proposed, when using the components, user can experiment with different component and network configurations. Based on experiments, a set of typical shape extraction scenarios is given, along with examples of respective data-flows. Graphic user interface is described that allows user to configure component networks and review their outputs. Finally, the system is compared to other applications, both commercial and academic.

2 IVP Framework

Image and Vector Processing Framework[8] was implemented to offer highly configurable shape extraction functionality. It's based on some straightforward observations: We have a set of “resources” at our disposal in the shape extraction process. Generally speaking, each resource fits into one of the two following categories:

- Objects and data structures (such as images, gradient maps, polylines/polygons, ...)
- Image and vector processing algorithms (image filters, edge detection, thinning, vectorization, ...)

Algorithms can be thought of as black boxes; we are in fact not at all interested in how they do their work, but only in the produced results. With this way of reasoning, the whole shape extraction process reduces to nothing more than a data-flow composed of algorithms that share and exchange processed data (see Figure 2.2 for an example of such data-flow).

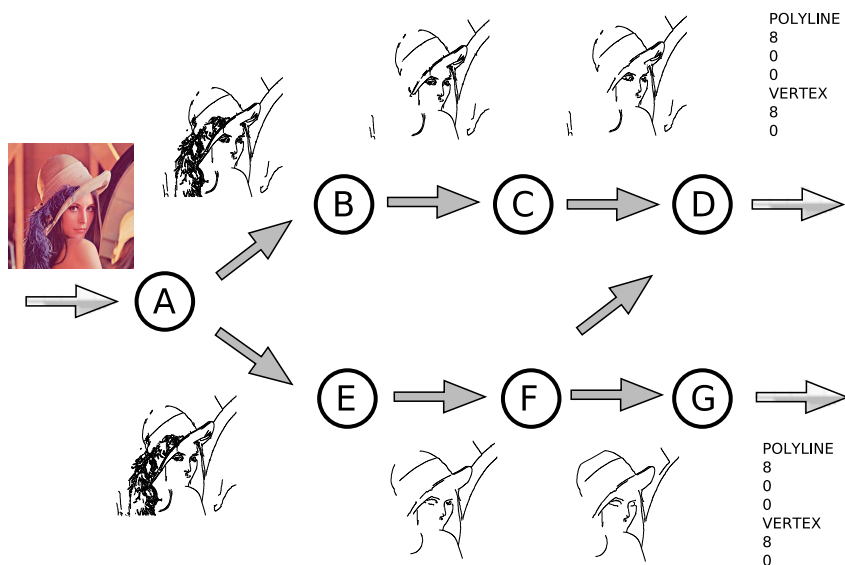


Figure 2.2: Algorithms as black boxes: Imaginary algorithms A-G form a data-flow with the arrows representing their *input* $\rightarrow$ *output* dependencies. A is an input algorithm that vectorizes the input image into a set of line primitives, B-C and E-F-G are algorithm branches that transform their vector input somehow. Algorithm D takes its two inputs and chooses one with respect to certain criteria. Finally, both D and G algorithms save their output in a specified format (file, stream, ...).

Image and Vector Processing Framework utilizes this perspective of view and provides component-based shape extraction environment, with each component representing one particular algorithm or a group of similar algorithms. Components

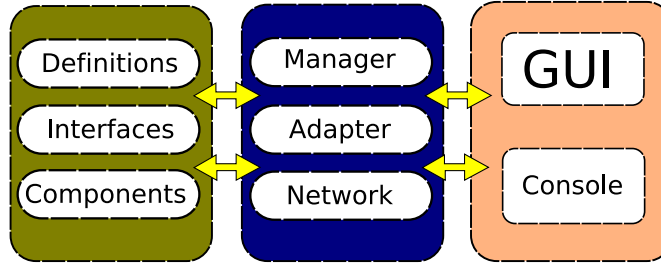


Figure 2.3: A diagram showing IVP Framework’s Component, Network and Application layers.

can define their inputs and outputs (as input/output ports); consecutively, they are interconnected to form a component network.

Component network represents the high-level shape extraction functionality and can be run on the demanded data.

Compared to specialized applications, this approach seems to be very flexible as the user can alter the output of a component network several ways:

- By changing configuration of certain components
- By changing component dependencies
- By using different components

Image and Vector Processing Framework is implemented under **.NET Framework 2.0**, with the *C++/CLI* being chosen as the main programming language (this hardly matters though because all .NET languages have roughly the same expressive strength; potential programmers can still develop components in the language of their own taste). All compiled code is fully verifiable (i.e. there is no unsafe or unmanaged code within the framework).

In this section, we will pass through all the framework layers (as depicted in Figure 2.3) and describe their function as well as give some handy implementation details for potential component developers.

2.1 Component Layer

2.1.1 Primitive Types & Metadata

We need to establish some basic primitive types and framework rules first; it would be otherwise impossible to agree on definitions of both processed objects and components later on.

Primitive types include structures to represent image and vector points, colors and connectivity information. For *C++/CLI* users¹, there is an additional list of typedefs to facilitate work with built-in .NET types (such as different precisions of integers, floats, ...).

¹languages such as C# don’t have any direct equivalent of typedef construction

.NET offers the possibility to add metadata into its assemblies; a set of metadata attributes is defined to further describe components, enable port registration etc.

Concretely, the resulting `definitions.dll` library includes code gathered from

- `attributes.h` - metadata attributes definition
- `config.h` - *C++/CLI* typedefs
- `definitions.h` - framework's primitive types
- `editors.h` - general set of `UITypeEditors` used to view component properties in a `PropertyGrid` control²

2.1.2 Interfaces

Objects and data structures that figure in the shape extraction process are abstracted using .NET interfaces and stored in `interfaces.dll` library. Each interface design is purely output-oriented which corresponds with the *input* $\rightarrow$ *output* data propagation within a network. Additional demand of transparency and simplicity (low-level access) is made on the interfaces to assure that they are usable by every thinkable component. For bitmaps, the choice of interface representation is usually simple, the problem becomes harder for vector data and will therefore be discussed in a separate chapter. A simple example of a bitmap interface is given in Listing 2.1.

Listing 2.1: `IGradientMap` interface example.

```
interface class IGradientMap {
    maxsize GradientMapWidth();
    maxsize GradientMapHeight();

    GradientMagnitude GetMagnitude(maxsize y, maxsize x);

    bool IsDir0(maxsize y, maxsize x);
    bool IsDir45(maxsize y, maxsize x);
    bool IsDir90(maxsize y, maxsize x);
    bool IsDir135(maxsize y, maxsize x);
}
```

There are two special interfaces that don't match any data structure or processed object. The first is the `IThumbnail` interface. Especially in the GUI, we often need to review the quality of a component's output. `IThumbnail` interface serves as a route through which components pass their thumbnail data (as a .NET bitmap or as an array of `GraphicsPath` objects in case of line drawings). The second extraordinary interface is the `IIVPFComponent` interface which is the obligatory parent interface for all framework components.

`Interfaces.dll` also contains definitions of three delegates that play an important role in port and thumbnail system implementation (those are shown in Listing 2.2).

²we are referring to `System::Windows::Forms::PropertyGrid` control

Listing 2.2: Important IVP Framework’s delegates.

```
Object^ GetInputInterfaceEventHandler()  
SetOutputInterfaceEventHandler(Object^ inter)  
SetThumbnailEventHandler(IThumbnail^ thumb)
```

2.1.3 Components

Components are encapsulated in managed .NET classes and loaded dynamically from .dll files by the means of .NET reflection (currently, all components are held in `components.dll` library). Each component inherits from `IIVPFComponent` interface which enforces basic component functionality (the interface is shown in Listing 2.3).

Listing 2.3: `IIVPFComponent` interface.

```
public interface class IIVPFComponent {  
    public:  
        void Initialize();  
  
        void Work();  
}
```

`Initialize` function serves two purposes: within it, default parameter initialization is recommended and moreover, its body is the only allowed place for dynamic port registration. `Work` function runs the component’s algorithm.

Component-specific parameters are implemented using .NET property system. Each public property of the class is considered to be a parameter and handled respectively. In order to display properties in a `PropertyGrid` control, parameters can be further described by the built-in .NET attributes (such as `CategoryAttribute`, `DescriptionAttribute`, ...). Each parameter of non-trivial type must also define a corresponding type converter.

As already mentioned, components exchange data by employing the concept of input and output ports. Both types of ports are identified with an unique name and must specify appropriate data-type (with respect to interfaces defined in `interfaces.dll`). Input ports can be mandatory or non-mandatory, while the output ports can restrict the number of connections to one at most - although this option is currently not used by any component, one can easily imagine situations (such as sequential data output components) when it’s essential.

Ports are modeled using .NET events, with the whole registration process having two stages - static and dynamic. To create a port, one must declare a corresponding public event in the class, either as an instance of `GetInputInterfaceEventHandler` for input ports or as an instance of `SetOutputInterfaceEventHandler` for output ports. Such event must be also properly decorated with one of the attributes given in Listing 2.4.

Attributes are needed to be able to obtain basic port information without instantiating the class first. This finishes the static registration phase. To truly export the output data, component needs to expose them using defined interfaces. One

Listing 2.4: Attributes used in static port registration.

```
InputPort[String^ name, Type^ type, bool mandatory]
OutputPort[String^ name, Type^ type, bool oneConnection]
```

Listing 2.5: An example of a component's class declaration

```
[ComponentDescription("...")]
public ref class GradientOperator
: public IIVPFComponent {
public:
    static enum class OpType {Sobel, Prewitt, Roberts};

    virtual void Initialize() override;

    virtual void Work() override;

    [InputPort("i1", IIRGBImage::typeid, true)]
    event GetInputInterfaceEventHandler^ i1;

    [OutputPort("o1", IGradientMap::typeid, false)]
    event SetOutputInterfaceEventHandler^ o1;

    event SetThumbnailEventHandler^ SetThumbnail;

    [Category("Configuration")]
    [Description("Gradient operator type.")]
    property OpType OperatorType;

    //...
};
```

possibility would be to implement these interfaces in the component's class, but that's rather inadvisable - aside from complicating class structure, we're heading for big code redundancy here as we can see that the input and output types are often the same in multiple components. That's where dynamic port registration steps in. Fixed set of data containers is defined (in `containers.dll`), each container implementing certain interfaces (these are data structures for images, gradient maps, line drawings, ...). In the `Initialize` function, it's possible (and required too) to tie each port with an object that implements the specified interface, which is one of the supplied containers or even the class itself. The dynamic dynamic port registration is accomplished by firing respective `SetOutputInterfaceEventHandler` events. In the `Work` function, we can query framework to give us inputs by firing input ports' `GetInputInterfaceEventHandler` events. A sample component declaration is shown in Listing 2.5.

2.2 Network Layer

The high-level functionality is contained within `core.dll` library which encompasses the means to identify and categorize components, connect components into networks and run the networks. The incorporated functionality is structured into three parts (each part being implemented in a separate class of the same name):

- **Component Manager** - handles components' categorization, instantiation, ...
- **Component Adapter** - wraps instances of components
- **Component Network** - handles network creation and execution

2.2.1 Component Manager

Component manager is the heart of the whole framework as it offers the most basic component handling capabilities. First of all, component manager loads components from `.dll` files using .NET reflection. When processing input `.dll` library, it generates the list of all its public classes. Each class is checked for `IIVPFComponent` interface compatibility; identified components are added to a component collection. Such loading and identification approach guarantees smooth processing even of whole directories, with no previous knowledge of their contents. Each added component is roughly inspected in order to get some basic component's description as well as port information. Only here we can fully appreciate the idea of static port registration that enables us to read port information directly with no need to create component's instance first. Once we obtain full list of components, further processing is required to identify sets of mutually compatible components. Considering their ports, components are also categorized into sets of

- Pure input components - components having no output ports
- Pure output components - components having no input ports
- Worker components - components having both input and output ports

The component manager then exports the gathered information about components through class properties.

Another component manager's duty is to care about components' instantiation (which cannot be done elsewhere), serialization and deserialization.

2.2.2 Component Adapter

Obviously, `IIVPFComponent` interface enforces the component classes only to contain minimal framework-related information. To connect components within a component network, each component is wrapped within a component adapter. The adapter provides each component with a unique string identifier (representing component's name) and creates data structures to represent component's port connections as well as component's current state. The allowed component states are

- **Invalidated** - we cannot guess anything about the component's inner state, the component's outputs are unreliable or inaccessible
- **Working** - component is running its algorithm at the moment
- **Ready** - component has finished executing successfully, so we can rely on its outputs

2.2.3 Component Network

Component network class administrates inter-component connections and component network execution (consult Figure 2.4 for an example of component's life cycle). Components can be added to or removed from a network freely as long as abiding some simple rules (e.g. components must be identified uniquely, component addition is handled solely by the component manager, each component is wrapped within a component adapter etc.). Ports of existing components are connected together pairwise (that means each connection is uniquely described by two pairs of component and port identifiers).

To execute the whole network, components are went through, starting from pure output components, advancing to pure input components; an alternative is to perform a local network update, which instead of executing all components at once takes one component and starting from it, it propagates changes through the rest of the network. Prior to network execution, all respective components are invalidated. During the execution, there might exist components that can't carry on with their work simply because they are never supplied their inputs. Such components are ignored and left invalidated. The execution stops at the moment when there are no components left to run.

Execution of the whole network usually takes some time; still, we would like to be informed about network's progress. Three events are provided to supply progress information: one event informs the subscribers about changes in components' states, another event fires at the moment when the network finishes its work. It might also happen that an exception gets thrown somewhere in the execution process (namely when you set components with incorrect parameters, such as invalid input or output file path). Upon catching an exception, special event is fired that returns exception description and the network execution is halted.

2.3 XML Configuration

In order to serialize and deserialize component network's configuration, one needs to choose proper data format first. Such format must have some fundamental properties:

- It should be plaintext-oriented to support easy user modifications.
- It should be transparent and well-structured.
- It would be wise to use some existing format for which .NET contains built-in parsing support.

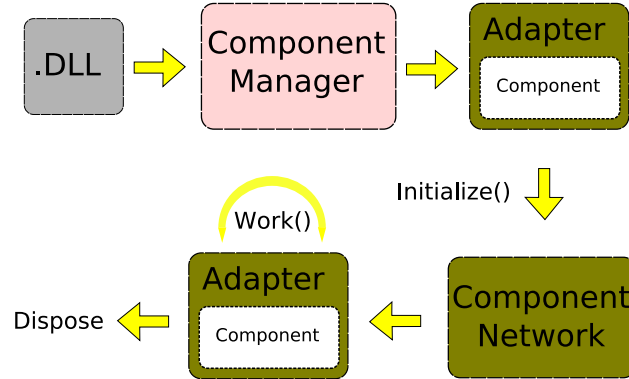


Figure 2.4: Diagram showing component's life cycle. First, the component is created by the Component Manager and wrapped within a Component Adapter. Being added to a Component Network, it is initialized first - then it's ready to work indefinitely until it's removed from the network.

In the IVP Framework, the decision was made to use XML documents. Although it's possible to give an exact format description using DTD, we will get along with just intuitive description and examples.

The respective XML document is divided into three parts. The first part (contained within the `<components>` element) contains component-specific information. Each component name and type is encapsulated in a `<component>` element. Component parameters are stored within `<parameter>` tags with their names and values; for properties of composite types, these tags can be nested within each other.

Another part of the XML document (contained within the `<connections>` element) reflects component connections. Each connection between a pair of components is stored in a `<connection>` element.

The last remaining part of the XML document (contained within the `<gui>` element) supports GUI network edition and is not obligatory. It contains information about GUI designer, components' locations and thumbnail settings.

An example of such XML document is given in **Appendix A**.

2.4 Application Level

Two applications are implemented that make the best of IVP Framework's capabilities: a simple console application that accepts a XML document and executes the respective network is accompanied by a full-blown GUI application which serves as a mean to design, run and evaluate component networks.

2.5 Problems Related To Framework Design

2.5.1 Component Parametrization

As mentioned before, IVP Framework uses .NET properties to implement components' parametrization. Properties in .NET are in fact nothing more but a syntactic sugar: each property field is allowed to specify its corresponding accessor and setter methods. We are mostly not interested in property access (access scenarios are simple and reduce to just reading parameters within a component or storing parameter values into a XML document). Setting a property is a trickier operation not only because we have to tackle input parsing but also because we must respect demands made on property integrity (i.e. among the set of possible datatype values, certain values are forbidden to use).

Fortunately, when it comes to parameter parsing, .NET gives us some strong means to get it done. Primitive types implement their own conversion routines to convert them to or from a string. Properties of composite types are inspected recursively using .NET reflection and only leaf primitive subproperties get parsed. In order to display composite property types in a `PropertyGrid` control, .NET concept of type converters is employed (i.e. each property field of composite type must be decorated with a proper `TypeConverter` attribute, otherwise it won't be editable in the `PropertyGrid` control).

The situation is worse when it comes to enforcing parameters' integrity. We can seldom find a datatype that directly fits the parameter's range of values - on the contrary, parameter types are often complicated (the most frequent non-trivial types are ranges and intervals, such as 0-100%, 0.0- ∞ , ...). .NET doesn't have any built-in mechanism to enforce these integrity requirements. Therefore, IVP Framework uses simple exception-based approach to satisfy the requirements: each time we pass an illegal value to a property setter, exception gets thrown which informs about unexpected error. This approach works well with the `PropertyGrid` control, still it has one big disadvantage - we are left no mean to guess legal parameter values in advance. One of the advisable work-arounds would be to decorate each property with some framework-specific metadata; but for this approach to work well, we would also need to write an equivalent of the `PropertyGrid` control that respects such metadata from scratch. Such component could then be highly user-friendly (i.e. displaying sliders or progress bars where needed, which is impossible unless you have precise type information). The implementation is however time consuming and the whole matter is only supplementary to the main framework's functionality; that's why it ended up on the to-do list.

2.5.2 Thumbnail Viewing

Thumbnail viewing represents the first step to create a collection of user-friendly viewing and editing tools. The thumbnail viewing mechanism is but a compromise between speed and desired functionality. The problem lies in the fact that a component is by no means limited in the way of storing its internal data. Hence, once we have to consider arbitrarily complex representations even of simple data, creating thumbnail from such data can be costly. We could of course specify a set of rules for data representation, but that would often degrade component's performance. The

chosen approach is natural: let components choose their interior data representation although it implies slow and memory-consuming thumbnail generation.

Through `SetThumbnailEventHandler`, each component is allowed to register one object as its thumbnails creator. Using the `IThumbnail` interface, thumbnail data are generated; the two currently supported data formats are .NET bitmap and an array of `GraphicsPath` objects for line drawings. In the GUI, the obtained data are saved and processed (small thumbnail is generated respecting the aspect ratio, full view is also available whenever required). The biggest performance bottleneck lies in thumbnail generation of bitmap objects - there is generally no other choice how to generate the thumbnail bitmap but to set each pixel's color separately; considering how slow it is to set separate pixels, generating thumbnails this way often takes ages.

2.5.3 Line Drawing Representation & Handling

Although it's usually simple to design interfaces and data structures for bitmaps, the situation complicates quickly with line drawings. Lets make some initial agreement: line drawing is composed of polylines and polygons (which we will call line primitives or just primitives), both considered to be "infinitely thin". Polylines and polygons are defined by a sequential list of points in R^2 space (in a polygon, the first and the last point in the list are automatically thought to be connected). We got to the first problem already: how should we pass the polylines and polygons through an interface? Should we use some high level structure or rely on low-level access? Especially for polygons, representation using a list of points can be misleading (although the list has its start and its end, polygon's points are naturally equal considering their position on the polygon).

Another problem arises when considering connectivity of particular primitives and the overall line drawing topology. For two connected line primitives, possible approach comes to mind to unify their endpoints so we can obtain connectivity information by comparing them. This approach is unfortunately based on if-then-else testing; having a set of primitives, we would have to perform the comparison for every pair of primitives separately, getting time complexity of $O(n^2)$ (and this computation would be necessary in each component that works with connectivity). IVP Framework takes a different path: additional information about connectivity is attached to the line drawing; hence, components that work with connectivity information have it at their disposal immediately. The question still remains how to represent the connectivity information - what complicates the situation is that polylines and polygons can be connected to other primitives in any of their points. Therefore, IVP Framework defines a set of constraints first for line drawing representation:

- A polyline is connected to other primitives merely in its endpoints.
- A polygon is connected to other primitives at most in one point; this point is the first point in the list.

Aside from its simplicity, this representation has some nice useful properties (e.g. when it comes to do polygonal approximation, there's no need to find maximal unconnected primitives, those are naturally contained in the input). Care must be

Listing 2.6: ILineDrawing interface

```

public interface class ILineDrawing {
public:
    maxsize NumberOfPrimitives();
    maxsize NumberOfPoints(maxsize primitive);
    bool IsPolygon(maxsize primitive);

    VectorPoint GetPoint(maxsize primitive, maxsize point);

    maxsize NumberOfConnectionPoints();
    ConnectivityInfo ConnectionPoints(maxsize primitive);
};

```

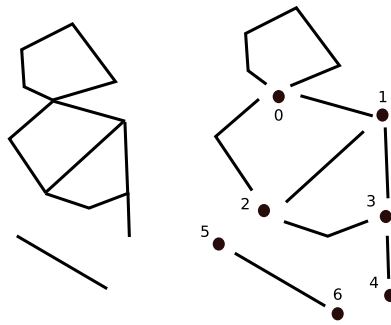


Figure 2.5: Line drawing along with its respective representation within the IVP Framework.

taken to avoid degenerate cases (such as two mutually interconnected polylines), which although being correct severely hamper further processing.

It remains to say how to append connectivity information to each primitive. This is done by designating each connectivity point with a natural number. The sequence of connection numbers should contain no holes to allow fast and straightforward indexing (i.e. with n being the number of connection points, we designate each connection point with a natural number from $\{0, 1, \dots, n - 1\}$ set). Each line primitive is assigned a pair of such connection numbers (for polygons, those two numbers are equal) as depicted in Figure 2.5.

Note that an unconnected primitive still consumes two connectivity points (the worst case is that for m primitives, we produce $2m$ connectivity points). An idea crosses mind to assign all unconnected endpoints to only one connectivity point (0, for instance) - that is of course possible and would work well, it however makes respective algorithms more complicated. The resulting interface is shown in Listing 2.6.

Containers.dll library contains some data structures to cope with line drawings and/or connectivity information. The most important are

- **CAssoc** - data structure used to logically merge primitives together into bigger entities

- **CConnectivity** - data structure used to handle connectivity information, connection point renaming, artifact identification, ...
- **CLineDrawing** - data structure that implements **ILineDrawing** interface, designed to hold all line drawing data

CConnectivity data structure is the most notable of these three. It uses connectivity information to effectively construct graph-based line drawing representation which can be successively used to find artifacts or just temporarily hold connectivity information; when it comes to removing primitives, it modifies the connectivity information accordingly and decides whether its time to join some of the remaining primitives together (to prevent degeneration).

CLineDrawing provides the basic line drawing representation capabilities, with functions to add primitives, specify their connectivity etc. A typical scenario is that one obtains a line drawing input, processes it somehow (using **CConnectivity** and **CAssoc** data structures) and saves it to a **CLineDrawing** container which is binded to one of the component's output ports.

2.6 Framework Discussion

The goal of this section is to compare approaches employed by the framework to different methods as well as to give some basic overview of framework's strengths and weaknesses.

2.6.1 IVP Framework vs. Specialized Applications

Compared to specialized applications, IVP Framework offers great room for experimentation, either with inter-component connections or with component-specific configuration; this is also its greatest strength, along with maximum existing code reusability. A domain where specialized applications are hardly beaten is the quality of produced outputs - the goal of the IVP Framework is to find and polish sound shape extraction methods, it cannot compete with a finely tuned application, especially with that which is aimed to handle just a subset of the shape extraction problem. Running algorithms in a component-based environment also necessarily implies some performance impact as well as increased memory consumption.

2.6.2 Interfaces

Modeling objects using interfaces is an applicable approach, still it's far from being ideal. Even if the components agree on the exchanged input/output data, they can hardly agree on their representation. The interface representation is however fixed and it might fit some components more than others; sometimes, component might need to develop significant effort to adapt the input data to its needs. It's also easy to imagine situations when this approach completely fails. For example, certain algorithms may need data with additional information which is not present in the existing interfaces (e.g. line thickness or color in line drawings) and because its impossible to alter the original interfaces, the only remaining way is to use component ports somehow. Lets follow the idea of colors here, where can we get them? Probably

where we extracted the former line drawing. Will we use the same algorithm to extract line primitives or a different one? In both cases we have problem: either with code redundancy or with our inability to match colors with the primitives coming through another port.

2.6.3 Multiple Object Processing

The framework in its current form doesn't support multiple input port connections, that means it cannot handle input data of variable size. This feature is simple to add though, it hasn't been implemented yet for two reasons: none of the currently implemented components requires it and it's questionable whether it's needed at all; it might be favourable to use some layer concept instead (i.e. each input is composed of several separate layers).

3 Component Catalogue

This section gives an overview of components already implemented in the IVP Framework. Those are mostly components designed to provide basic shape extraction functionality and to demonstrate framework potential. All the proposed components have no more than one input and one output port, port types will be denoted by simplified notation `input type` $\rightarrow$ `component name` $\rightarrow$ `output type`.

Algorithm description follows where needed, along with examples of *input* $\rightarrow$ *output* transformations. Component parametrization is mostly related to the described algorithm; hence, parameter specifications are omitted (user can familiarize himself with the parameters easily when using framework's GUI).

3.1 Image Processing Components

The first group of components aims to provide digital image processing functionality, including image input/output, resampling, thresholding and smoothing.

Image Load/Save Components

`$file` $\rightarrow$ `ImageFromFile` $\rightarrow$ `RGB Image`

`RGB Image` $\rightarrow$ `ImageToFile` $\rightarrow$ `$file`

Loading image into the network is usually the first step in the majority of shape extraction tasks. The ability to store images too comes handy when preprocessing large image sets or simply when there is need to use implemented image processing filters alone.

Image Resample Component

`RGB Image` $\rightarrow$ `ImageResize` $\rightarrow$ `RGB Image`

Input images might contain sampling defects (they might be too large or have unacceptably different sizes, ...). Resizing can be done absolutely (by specifying desired output size) or proportionally. The component tackles resizing by implementing Nearest pixel, Bilinear And Biquadratic interpolation algorithms.

Nearest pixel resampling is the most straightforward - for each pixel of the output, the corresponding input pixel is calculated using trivial scaling formula; for big scale differences, this approach works rather poorly, producing substantially pixelized images.

Bilinear resampling takes 4 pixels to compute the resulting color: first, destination point's location is recomputed into input image coordinates. Its 4 adjacent pixels are averaged using distance ratios between the recomputed point and the real input image pixels (as seen in Figure 3.2).

Biquadratic resampling is even gentler method that works with a 3x3 matrix of adjacent pixels. These 9 pixels are divided into triplets and interpolated using quadratic functions. Biquadratic resampling usually generates the nicest results of all the mentioned methods.

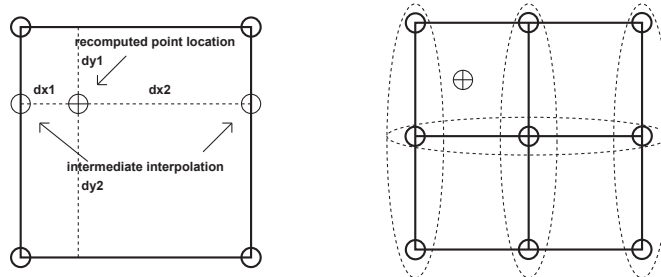


Figure 3.2: The first picture demonstrates bilinear interpolation principle, while the second picture shows pixel triplets considered by the biquadratic interpolation algorithm.



Figure 3.3: Thresholding example. Essential shape information is extracted by filtering black color layer from the input picture.

Gaussian Smoothing Component

`RGB Image` $\rightarrow$ `GaussianFilter` $\rightarrow$ `RGB Image`

Processing of noisy images may require a preliminary smoothing step. For its properties, Gaussian smoothing has become a well-established method (even as a part of certain edge detection algorithms). The component allows configuration of both the σ parameter and the window size to obtain the desired smoothing effect.

Image Thresholding Component

`RGB Image` $\rightarrow$ `RGBThresholding` $\rightarrow$ `Binary Image`

In certain situations (such as cartoon drawings, maps, low-quality pictures), plain thresholding still represents a robust and credible way to extract demanded features from the input image (see Figure 3.3). The component allows user to specify a linear range of colors corresponding with image foreground, rest of the pixels is ignored as background.

3.2 Edge Detection Components

Edge detection is a crucial step when extracting important shape features. The present implementation relies for the most part on first derivative edge detection methods, particularly on a set of methods which are often together denoted as the

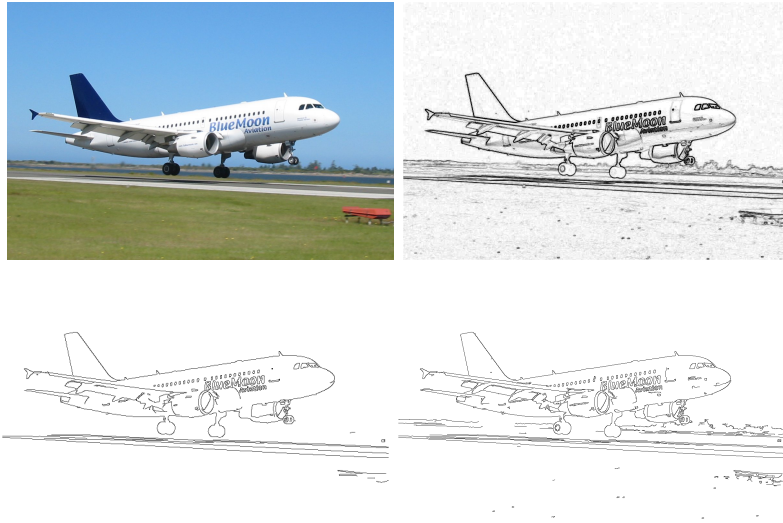


Figure 3.4: Edge detection example. First picture shows the input image, second picture displays calculated gradient map. Bottom pictures both represent detected edges (with *upper* = 30, *lower* = 10 and *upper* = 20, *lower* = 5 thresholds).

Canny operator[6]. The whole first derivative edge detection process comprises multiple stages, including

1. Image smoothing
2. Gradient approximation using first derivative operators
3. Non-maxima suppression
4. Hysteresis thresholding or other approach to label edge pixels in the gradient map

Each stage is handled by an individual component to support future extensions (consult Figure 3.4 for examples).

Gradient Operator Component

`RGB Image` $\rightarrow$ `GradientOperator` $\rightarrow$ `Gradient Map`

Prior to edge detection, gradient map needs to be computed from input image pixel intensities. This is accomplished by applying first derivative operators (depicted in Figure 3.5) while shifting a 3x3 (in case of Sobel and Prewitt operators) or 2x2 (in case of Roberts Cross operator) window over each image pixel. Gradient magnitudes in X and Y directions are computed directly by the operator definition as a weighted sum of respective pixel intensities. From their values, we can guess approximated gradient directions (in 45 degree steps).

Non-Maxima Suppression Component

`Gradient Map` $\rightarrow$ `NonMaximaSuppression` $\rightarrow$ `Gradient Map`

1	2	1
0	0	0
-1	-2	-1

1	1	1
0	0	0
-1	-1	-1

1	-1
-1	1

Figure 3.5: First order derivative operators: Sobel, Prewitt and Roberts Cross.

There's nothing to prevent gradient map created by first derivative gradient operators from containing thick areas of high gradient magnitude. But having thick areas doesn't match our desire to identify edges; preferably, we would like the areas to be as thin as possible. Non-maxima suppression searches for peaks in gradient magnitude with respect to the gradient direction and ignores the pixels that lie on the gradient slope. Tracing the gradient map sequentially, all that is needed to be done is checking two adjoining pixels in the gradient direction - once they both have lesser magnitude, we found a gradient peak.

Hysteresis Thresholding Component

Gradient Map $\rightarrow$ HysteresisThresholding $\rightarrow$ Gradient Map

Hysteresis thresholding extracts edge pixels from a gradient map based on two thresholds. Each pixel with gradient magnitude greater than the upper threshold is marked as an edge pixel immediately. Pixels with gradient magnitude greater than the lower threshold are marked as candidate pixels. Candidate pixel is marked as edge pixel if and only if it lies on a candidate pixel chain which also contains at least one pixel with high gradient magnitude. This partially resolves a common problem when tracing edges: even important edges don't have high gradient magnitude everywhere. On the other hand, areas with medium or low gradient magnitude gain importance when connected to high magnitude areas. Practically, the pixels are marked using depth-first search on the gradient map.

Second Derivative Edge Detection Component

RGB Image $\rightarrow$ ZeroCrossings $\rightarrow$ Binary Image

Second derivative edge detection forms an alternative to the aforementioned methods. In general, it is much more sensitive to noise and false edge responses. Using a simple mask, the algorithm tries to roughly approximate the second derivative of intensities for each image point. Consecutively it identifies pixels where the second derivative crosses zero and marks them as edges. Use of threshold is preferred to filter doubtful responses.

3.3 Segmentation Components

Segmentation is usually defined as a procedure that takes an input image and divides it into contiguous areas, each area sharing some general properties (low-level,



Figure 3.6: Input image along with the segmentation result composed of 3 layers.

semantic, ...). Good image segmentation may also well preserve shape information, which is the reason for implementing following components.

Segmentation Component

`RGB Image` $\rightarrow$ `Segmentation` $\rightarrow$ `Segment Map`

The main goal of this component is to find an image segmentation approach to produce areas that well correspond with contained shapes (consult Figure 3.6 for an example). The component is presently driven by the K-means segmentation algorithm on feature vectors of RGB or RGBXY pixel values with euclidean distance.

K-means represent a general approach commonly used for object clustering. All we need for this method is a rule that binds an object to a defined feature vector and a measure to compare inter-object distances. Initially, a set of K cluster centers is chosen either randomly or with a proper heuristic. In each algorithm step, objects are assigned to their nearest center. Cluster centers are recomputed then as medians of respective assigned feature vectors. The procedure stops at the moment when no objects longer change centers.

Segmentation Vectorization Component

`Segment Map` $\rightarrow$ `SegVectorization` $\rightarrow$ `Line Drawing`

Contours of contiguous areas produced by the segmentation algorithm may or may not serve as useful shape descriptors. One straightforward approach would be to turn area contours into a set of polygons, which is in fact exactly what the component does. Care must be taken to ignore unreasonable contours (such as contours containing image borders that seldom carry useful shape information). The contiguous areas are identified with respect to 4-connectivity. Typical output is depicted in Figure 3.7.

3.4 Binary Image Processing Components

Binary images, despite their simplicity, play an important role in image processing tasks. Binary image can have different meanings depending on situation (foreground/background, object/background, important/unimportant features, ...) and



Figure 3.7: Vectorization of segments example. The input is segmented into 5 layers, contours of contiguous regions are extracted afterward.

when it comes to edge detection, it serves as a powerful mean of edge representation. Unless stated otherwise, in the following text we will stick to foreground/background semantics; by pixel neighbors we will mean adjacent foreground pixels only.

Morphologic Operator Component

Binary Image $\rightarrow$ ErosionDilatation $\rightarrow$ Binary Image

Morphologic operations are irreplaceable when handling binary image defects (such as gaps, curved objects borders,...). Each image pixel with its respective neighborhood is compared against a properly chosen mask. For a foreground pixel in the original image, dilatation operator marks neighboring pixels as foreground according to the mask. Erosion is the direct opposite, it preserves the foreground pixel only of its neighborhood fits the mask. Erosion and dilatation operations can be successfully applied in sequence, forming opening and closing operators.

Thinning Component

Binary Image $\rightarrow$ Thinning $\rightarrow$ Binary Image

Thinning algorithms are handy when you intend to polish results given by the edge detection or when there is need to find skeletons of thick objects.

Stentiford thinning algorithm[11] uses four erosion matrices. The image is went through for each mask and pixels that fit the mask are deleted (unless they are endpoints or connect more than one image component) until there remain no points left to remove.

Zhang-Suen approach[13] also comes with a set of erosion masks, combined with more complicated removal criteria based on the crossing number and the number of neighbors.

Figure 3.8 shows comparison of these two approaches.

There's not always need for full-fledged thinning process, we often know that the binary image contains reasonably thin lines and just want to remove sharp turns on them. Staircase removal algorithm inspects the image just once and smooths contained lines.

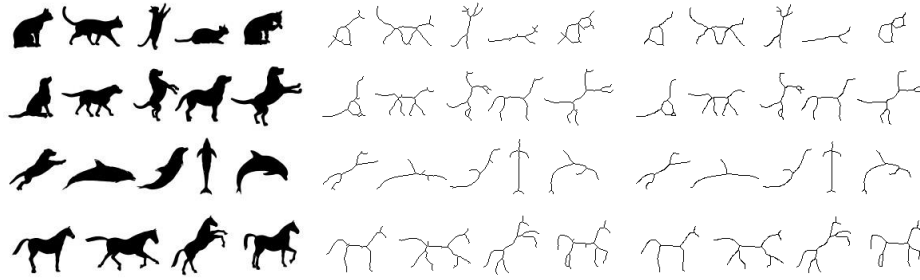


Figure 3.8: A comparison of Stentiford and Zhang-Suen thinning outputs. Zhang-Suen approach seems to produce smoother results with less artifacts.

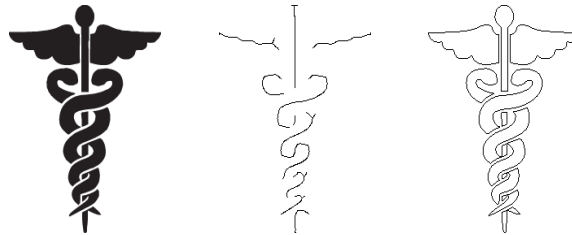


Figure 3.9: Input binary image along with the respective contouring and thinning output.

Contour Extraction Component

Binary Image $\rightarrow$ Contouring $\rightarrow$ Binary Image

Skeleton is not the only potential representation of a thick object, contour is often as much valuable (as seen in Figure 3.9). Contours are extracted with sequential image tracing as described by Ablameyko[4], although here we work with full image matrix instead of run-length encoding.

Vectorization Component

Binary Image $\rightarrow$ Vectorization $\rightarrow$ Line Drawing

Vectorization component turns binary image into a set of polylines or polygons, using the pixel-chain vectorization method found in [4] and [5]. It however investigates the problem from a slightly different perspective, using the concept of “pixel linking”. Consider an arbitrary foreground pixel, in fact we only need to decide which of the at most 8 neighbors should be linked to it. Once we have a consistent set of rules that link a center pixel to its neighbors, we can easily divide binary image into pixel chains between endpoints (points with 1 link only) and junctions (points with more than 2 links). These pixel chains are transformed into polylines immediately. One trouble remains with closed cycles within the image, hence an additional pass is needed to identify unprocessed pixels and turn those into polygons.

The following rule is employed for pixel linking (Figure 3.10): Link pixel to all of its 4-connected neighbors. Diagonally connected neighbor is linked to a center pixel

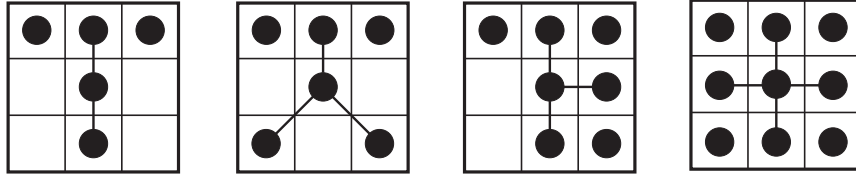


Figure 3.10: Model pixel linking situations. In the first place, center pixels are linked to 4-connected neighbors. Diagonal neighbors are considered next.

only if it's inaccessible from the center using 4-connectivity. For thick regions, this approach still produces well-defined (but degenerated) output. Moreover, the output is always intersection-free, which wouldn't hold if we trivially connected a pixel to all of its 8-connected neighbors.

Both polylines and polygons are stored along with connectivity information, which is almost impossible to reconstruct retroactively.

3.5 Vector Processing Components

Acquiring the vectorized shape form brings us halfway the shape extraction process. Obtained data are seldom satisfactory, more likely they appear as nothing more but a mess of short insignificant lines. Hardest step awaits - the attempt to reconstruct meaningful shapes.

Polyline Simplification Component

Line Drawing $\rightarrow$ PolylineSimplification $\rightarrow$ Line Drawing

For different reasons, polylines and polygons may suffer from various irregularities. It's also no exception that they contain much more information than required. Polyline simplification, while handling input defects, also greatly reduces the size of vector data. We can formulate the polyline approximation problem two ways

- *min* - ϵ problem - given the number of line segments, find the approximation with the minimal error
- *min* - $\#$ problem - given the maximum allowed error, find the approximation with the minimum number of line segments

In both categories, there exists plenty of approximation algorithms based on dynamic programming, graph pruning, finding the shortest path within a directed acyclic graph, $\dots$. The methods are either optimal (in relation to certain error measure, such as $L1, L2, L\infty, \dots$) or suboptimal. Optimal approaches mostly share the disadvantage of big time complexity (in order of $O(n^3)$) and for practical purposes, they present no substantial improvement over suboptimal methods.

Polyline approximation can be calculated for the whole line drawing or for each line primitive separately, the former being a fundamentally harder problem. The

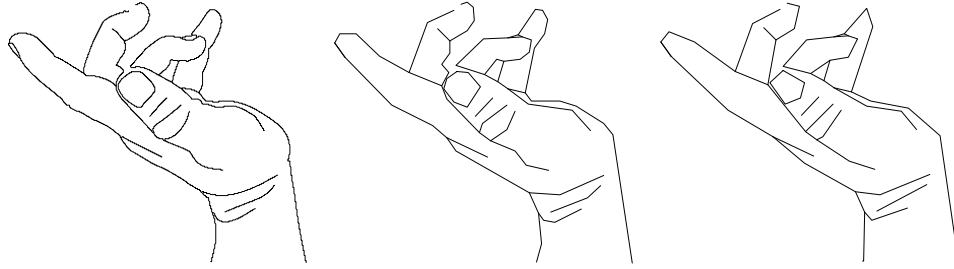


Figure 3.11: Polyline simplification sample. The input is simplified using the Rosin-West and Wall-Danielsson (with $\epsilon = 5$) algorithms. Rosin-West reduces the number of line-segments from 1610 to 84. Due to large tolerated error, Wall-Danielsson produces coarser approximation with only 63 line segments.

component therefore processes primitives one by one with suboptimal *min-#* methods.

Douglas-Peucker algorithm[7] for polyline simplification recursively splits the polyline in the point of maximum deviation from the approximating straight line segment as long as the approximation error exceeds some given threshold.

Rosin and West proposed a non-parametric modification of the same approach[9]. Series of recursive divisions define a binary tree, with each two sibling nodes representing a better approximation of the parent polyline. A measure of significance is defined as a ratio of the line segment length divided by the maximum deviation of the polyline from the approximating straight line segment. Based on the significance, the tree is traversed from leaves to the root. At each node, we compare the respective significance to its children. If its greater than both of them, parent approximation is kept and passed up in the tree; approximation represented by the children is used otherwise.

The method of Sklansky and Gonzales[10] inspects polyline points in their natural order, hence it has only linear time complexity. First point is retained, decision whether to keep the other points is made upon gradual construction of conical intersections; the last point to fit within the intersection is kept, the intermediate points are ignored.

Approximation approach published by Wall and Danielsson[12] merges points that satisfy certain criterion, which here is the deviation area between the polyline and the approximating line segment divided by polyline length.

Some approximation results are depicted in Figure 3.11.

Situation is a bit more complicated when it comes the polygons. Polygon simplification can be thought of as a separated problem; alternatively we can heuristically choose one initial point or line segment and handle the rest of the polygon with one of the preceding approaches (for example, with Sklansky-Gonzales and Wall-Danielsson methods, it's clever to start from the point farthest from the polygon's center of gravity). To keep things simple when utilizing the second approach, current implementation doesn't bother with potential degeneration and chooses starting conditions randomly (i.e. it always retains the line segment between the first and the last point in the list).

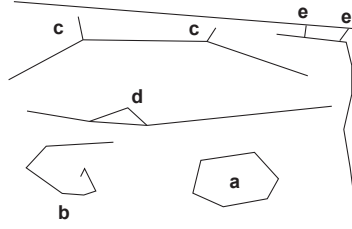


Figure 3.12: The most usual artifact types. **a.** short cycles **b.** unconnected polylines **c.** one-connected polyline artifacts **d.** spurious loops **e.** insignificant junction connections

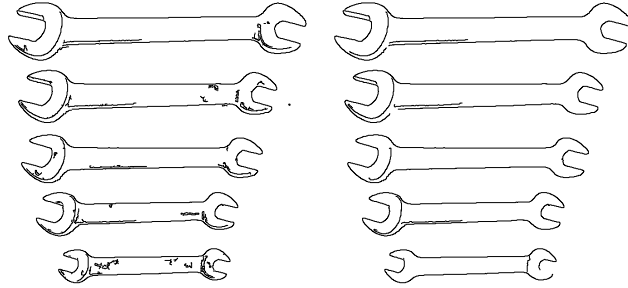


Figure 3.13: Artifact removal example. The former 349 primitives are cut down to just 18 with no loss of shape information.

Artifact Removal Component

Line Drawing $\rightarrow$ ArtifactRemoval $\rightarrow$ Line Drawing

Aside from polyline and polygon defects, input vector data may contain artifacts at higher logical level (i.e. with respect to topology and shape semantics). These include redundant polylines or polygons of negligible size, spurious loops or junction artifacts (for a complete list of distinguished artifacts, check Figure 3.12). All the mentioned faults usually share mutual characteristics: the involved polylines or polygons are short. Obviously, removing one artifact vector may uncover more input defects, hence an iterative algorithm is employed to correct input data as much as possible. Within one iteration, all identifiable artifacts are marked and removed instantaneously. The algorithm ends upon reaching a specified number of iterations or when there are no artifacts left to remove. Artifact identification is accomplished by checking connectivity of each primitive. Figure 3.13 demonstrates typical artifact removal situation.

Iterative Artifact Removal Component

Line Drawing $\rightarrow$ IterativePruning $\rightarrow$ Line Drawing

Artifact handling based on length is a sound policy, still it doesn't anyhow guarantee the output to be suitably small and compact. Choosing a specified number of the most important line primitives is a preferable way in almost all situations, and a very

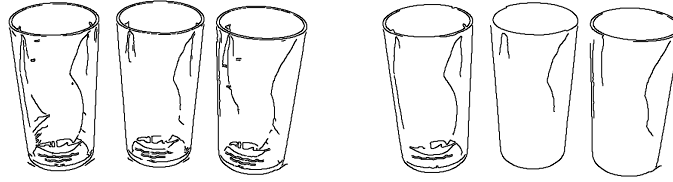


Figure 3.14: The first picture shows line drawing containing 386 line primitives. In the second picture, there is the result of iterative artifact pruning containing 33 primitives.

hard problem to solve too. With this component, an experimental path was taken to find out if it's a good idea to prune primitives based on their length. An upper bound on the desired number of primitives is selected by the user (we cannot work with exact numbers here, as removing one primitive might result in merging others). As a first step, primitives are sorted by their length in increasing order. The algorithm works iteratively and tends towards the specified number of vectors; in each iteration, we expect to eliminate certain number of primitives (the approximate number of primitives to eliminate can be specified in percents to affect the convergence speed, there's however no direct relationship between the convergence speed and the quality of the produced output). Elimination criterion is based on length (short primitives are thrown out first) and knowledge about artifacts (which is similar to that used in the `ArtifactRemoval` component). It might easily happen that we are unable to eliminate the desired number of primitives within one iteration. In such case, we eliminate ten percents of primitives blindly.

Despite the simplicity, produced results are often quite satisfying (as in Figure 3.14). The algorithm performs well as far as the input contains big connected shapes. On the other hand, it fails badly when processing largely unpreserved shapes with lots of noise.

Polyline Connection Component

`Line Drawing` $\rightarrow$ `PolylineConnection` $\rightarrow$ `Line Drawing`

Due to noise, imprecise edge detection or overlap, significant polylines might get divided into multiple disconnected parts. It is necessary to reconnect these parts again into whole pieces (this is achieved nicely in Figure 3.15). Component implements algorithm that is described in its extended form in Altman's thesis on map digitalization[5]. First of all, candidate polylines are identified in the input. These candidates are then iteratively connected until no further connections are possible. User is offered several criteria for polyline connection.

The easiest way to connect polylines is to specify maximum gap between their endpoints, which is usually only practicable for very small gaps (it generates too many errors otherwise).

Second criterion is based on polylines' orientation and gap sizes. Optional settings include intersection handling, the ability to ignore large gaps for collinear vectors as

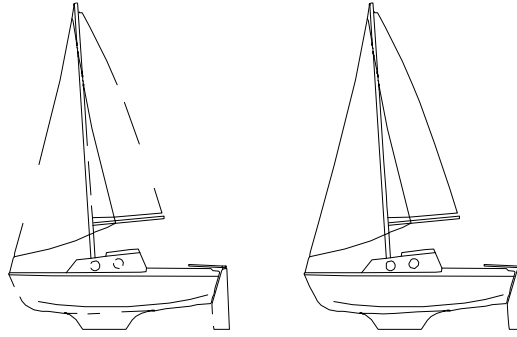


Figure 3.15: Ideal situation to connect polylines. Shapes are nicely preserved; hence, it's simple to reconnect polylines correctly.

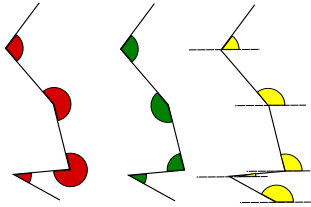


Figure 3.16: Different types of angle measuring: one side angles, smaller angles, X-axis angles.

well as bilateral connection check (i.e. polyline A is connected to polyline B only if polyline B agrees).

3.6 Line Drawing Output Components

Vector data need to be saved to a file for further processing or similarity matching, either in some well-known format or as time series.

Line Drawing Save Component

`Line Drawing` $\rightarrow$ `LDToFile` $\rightarrow$ `$file`

The component saves its input to a file using specified file format (DXF, textfile, ...).

Line Drawing Transformation Component

`Line Drawing` $\rightarrow$ `LDToAngles` $\rightarrow$ `$file`

A component transforming input data to time series for the sake of similarity matching. Each primitive is first normalized by splitting it into a given number of equally long line segments. The output representation is formed by the respective angles, which can be measured as angles on one arbitrary side of the polyline/polygon, the

smaller angles between each pair of consecutive line segments or as angles formed by the line segment and the X-axis (an example is given in Figure 3.16).

4 Domain Scenarios

Obtaining good vectorization results is impossible for large heterogenous image sets. On the contrary, as long as we work with distinguished collections of similar images, we can use the component-based framework to design typical scenarios, each scenario being aimed for one particular shape extraction situation.

It would be foolish to think that we can get along with the implemented set of components even when processing one particular image type: fixed network parametrization will still work for some images better than for others and certain inputs will get completely mishandled. The purpose of this section is to give an overview of the experimentation results on various types of images (drawings, photos, ...); for each image type, the example of an assembled network is given along with an example of a typical data-flow. Given networks are in no way comprehensive and could be improved by adding components; the goal of the examples is but to give an idea of applicable approaches.

4.1 Map

When processing maps, the interest usually lies in extraction of different map layers, those are point marks (buildings, orientation points, ...), line features (contour lines, roads, railroads, ...) or contiguous areas. Figure 4.2 demonstrates typical map processing network configuration.

In the example in Figure 4.3, we will try to extract contour lines and forest boundaries. Both features are extracted first using RGB thresholding. Contour lines need to be thinned for further processing, which also carries a necessary artifact removal step (artifacts are present thanks to thinning and also because there are marks in the map having the same color as contour lines). Contour lines in the original map are disconnected (this is a usual situation in almost all maps as there are extra symbols within the map that collide with contour lines); hence, we try to reconnect them consequently. Finally, polyline approximation step is undertaken to smooth the result.

Forest areas (visualized with dark green color) present a bigger challenge as they contain gaps and there are patterns within the image drawn with the same color. Upon extracting the respective color layer, dilatation is performed to close or at least reduce the gaps. The result is subsequently vectorized; thanks to the fact that the forest areas are relatively large, we can use **ArtifactRemoval** component to remove undesired objects (including the mentioned hatch patterns).

4.2 Black And White Drawing

As for the monochromatic drawings, the vectorization process is usually quite straightforward compared to other image types (see Figure 4.4 for an example network). We simply filter the black color layer by the means of thresholding; in case of noisy pictures, this step might require additional postprocessing by morphologic operators. Further, we must decide whether we want to extract drawing's contour or its skeleton and use appropriate components (i.e. **Thinning** component for skeletons, **Contouring** component for contours). For thick objects, obtained skeleton may be

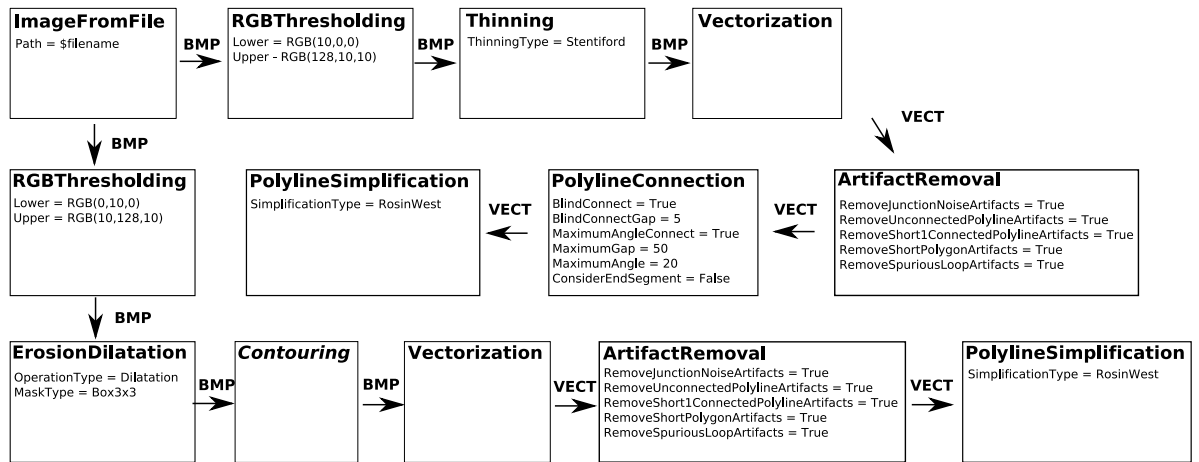


Figure 4.2: Scenario 1 - Map

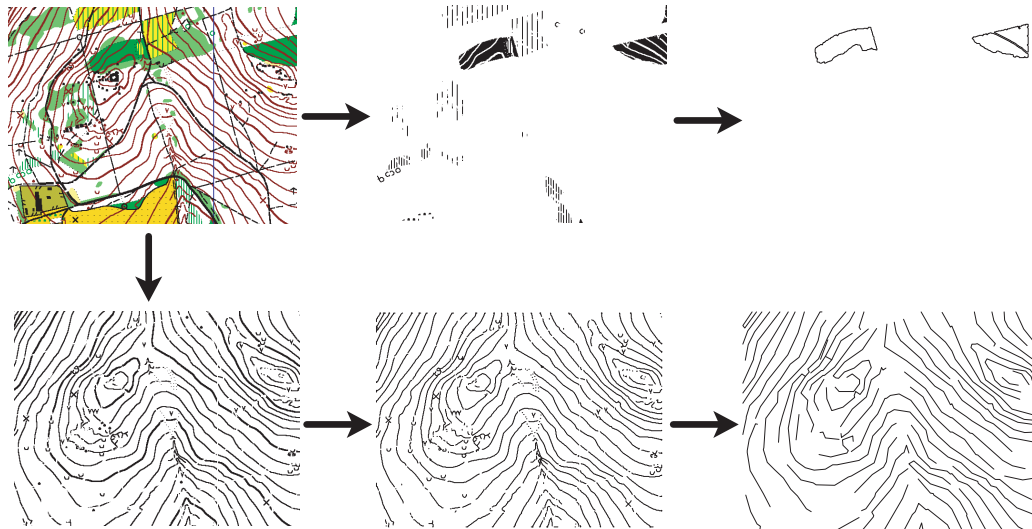


Figure 4.3: Scenario 1 - Map - Data flow

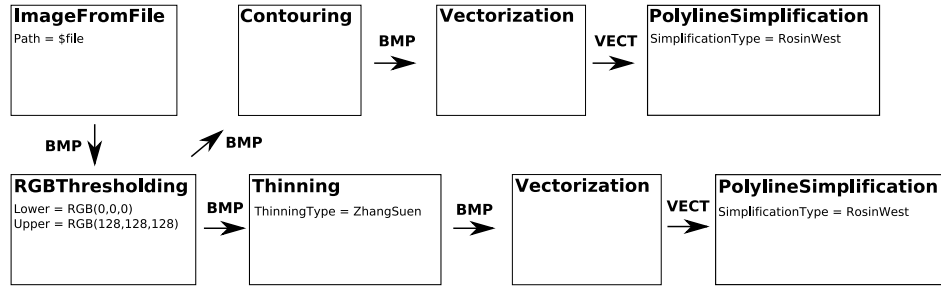


Figure 4.4: Scenario 2 - Black and white drawing

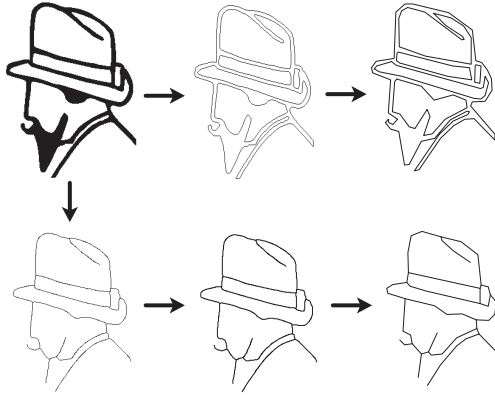


Figure 4.5: Scenario 2 - Black and white drawing - Data flow

imprecise and have poor quality; on the other hand, for thin objects (such as lines), contouring is unnecessary. Because of the intention to give a nice overview example, an image was chosen that is suitable for both skeleton and contour extraction (consult Figure 4.5).

4.3 Logo

Even though not as simple as monochromatic drawings, logo processing still introduces no unsurpassable difficulties. For most logos, we could use simple RGB thresholding to filter out important layers (text, border, interior, ...). Edge detection is however more credible here, as it works with local changes of image intensities and we need not bother to specify different color layers manually. The intensity contrasts are usually big within logos; sometimes, one must occupy himself with the setting of hysteresis thresholding bounds but this is not the case. The resulting vector output has principally very high quality. Still, we are interested in the removal of contained artifacts (typically, those are short one-connected polylines). A necessary step is to connect polylines together, even the best edge detection result contains disconnected corners and small gaps here and there. The respective network is depicted in Figure 4.6. In the given example, it suffices to connect polylines blindly; connecting polylines based on their orientation is also a smart choice here. Typical data flow example is given in Figure 4.7.

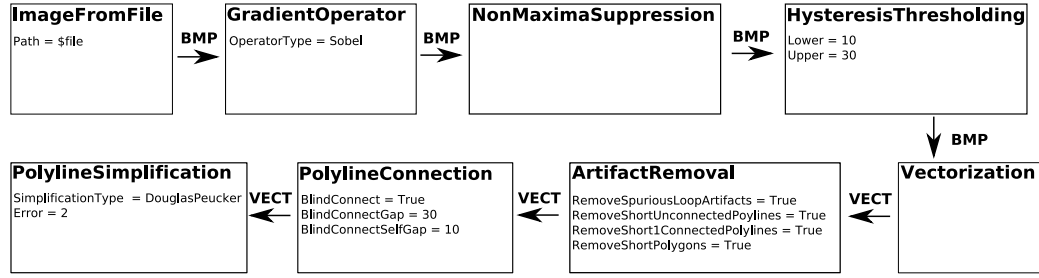


Figure 4.6: Scenario 3 - Logo

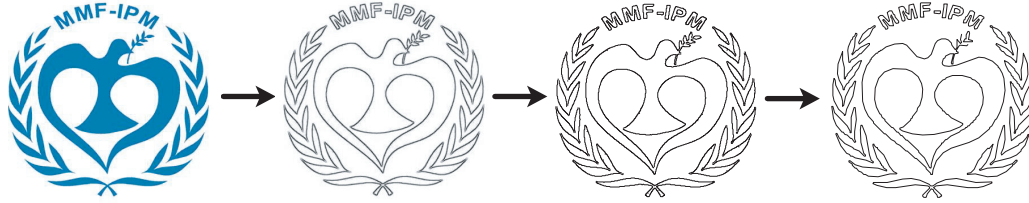


Figure 4.7: Scenario 3 - Logo - Data flow

4.4 High Contrast Object

In case of high contrast objects (i.e. when background is identified easily), we basically need to choose from two alternatives: We can use some segmentation algorithm to filter foreground from background or we can rely on edge detection. It's hard to say which of these two approaches is the better one, that rather depends on the particular input image. In the example in Figure 4.9, both ways are shown on a simple image. The output of segmentation algorithm is vectorized immediately and cleaned of artifacts (the only artifacts in such output are small uninteresting polygons), almost the same applies to the edge detection output. Once we have obtained artifact-free output, we can process it further with polyline simplification algorithms, PolylineConnection component etc. Check Figure 4.8 for the complete network.

4.5 Complex Object

There are image categories which IVP Framework currently has no chance of handling, those are mainly photos of complex scenes (nature, streets, buildings, cars, ...). The example given in this scenario is on the edge of current IVP Framework's capabilities. Here we are to tackle vectorization of one complex object, which is comprised of multiple parts, therefore it's not immediately apparent which of these parts should be kept and which removed. We are left no choice but to use edge detection in this case as both segmentation and thresholding approaches might fail easily. No matter what thresholds are chosen for the hysteresis thresholding step, the vectorized edge detection output always contains considerable amount of noise. The decision to use ArtifactRemoval component would be well justified, but as

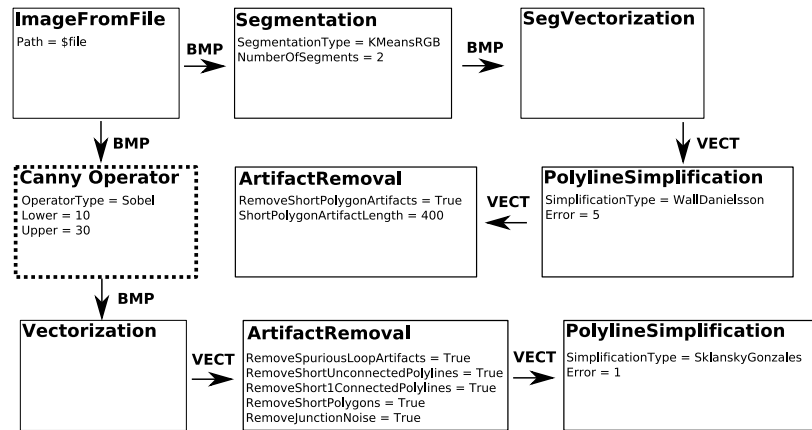


Figure 4.8: Scenario 4 - High contrast object

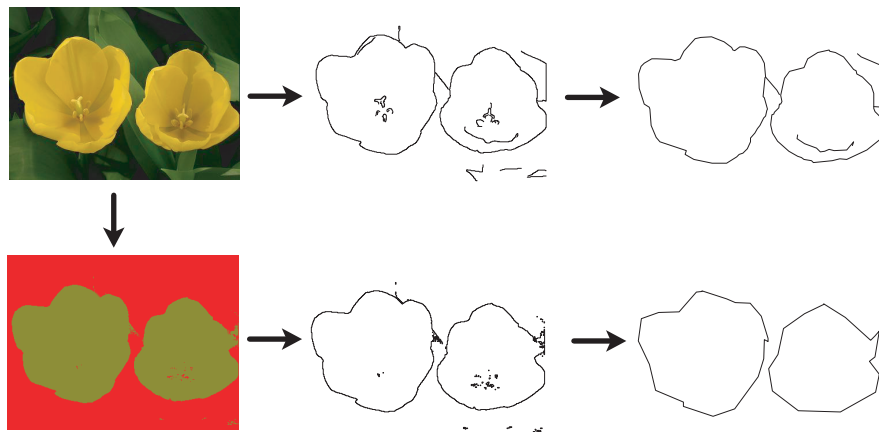


Figure 4.9: Scenario 4 - High contrast object - Data flow

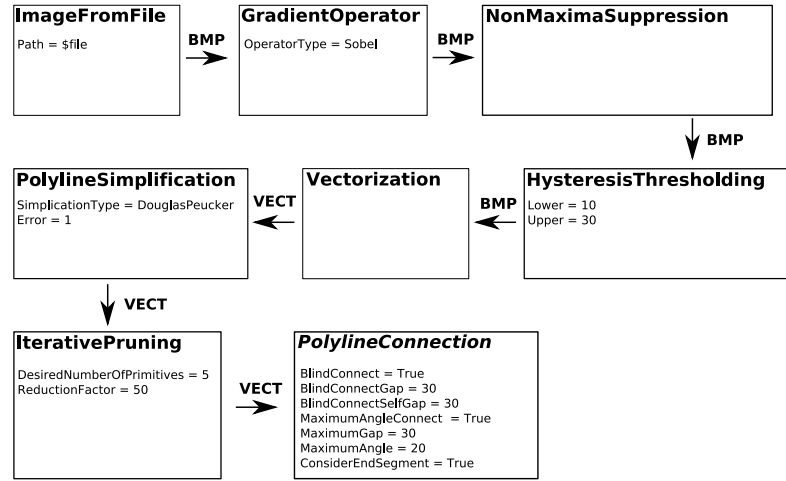


Figure 4.10: Scenario 5 - Complex object

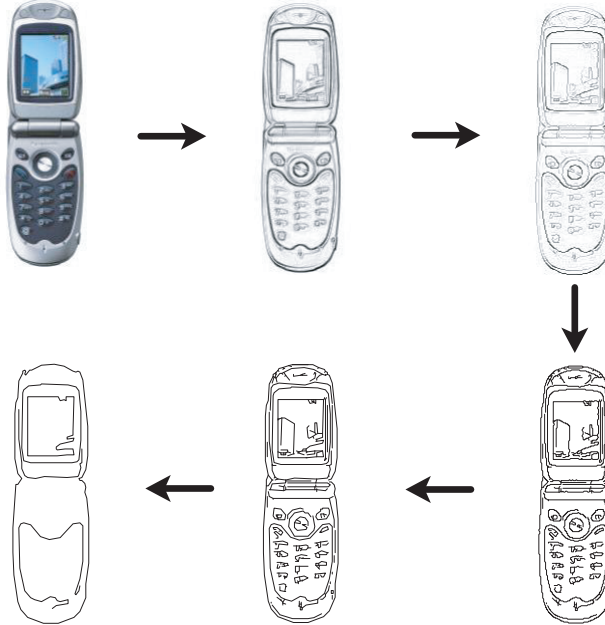


Figure 4.11: Scenario 5 - Complex object - Data flow

long as we don't know anything about the input, different approach comes to mind - we may try to use **IterativePruning** component to directly obtain the desired number of primitives. In the given example in Figure 4.11, this approach works well, in others it may fail. The result can be polished by merging disconnected polylines or by polyline simplification. Figure 4.10 shows the scheme of network configuration under this scenario.

5 User Manual

5.1 Requirements

- To run the application smoothly, *Microsoft Windows XP* are recommended. *Windows Vista* should run it too, yet without any guarantee.
- *Microsoft .NET Framework 2.0* (or any newer version containing 2.0 runtime) is required.
- System with 512 MB of RAM is required. To use full graphic user interface functionality, at least 1 GB of RAM is recommended.

5.2 Installation

Before installation, make sure that your system contains .NET Framework 2.0, else you must install it first by running `dotnetfx.exe`. In order to install the framework, run `setup.exe` file and follow given instructions. During the installation process, you can choose destination program path. Some of the installed files are listed below:

- **containers.dll** - containers and data structures
- **core.dll** - IVP Framework's high-level functionality
- **definitions.dll** - IVP Framework's basic declarations, types, attributes, ...
- **interfaces.dll** - interfaces that describe processed objects
- **ivpf-console.exe** - framework's console application
- **ivpf-gui.exe** - framework's graphic user interface
- **WeifenLuo.WinFormsUI.Docking.dll** - third party library required to implement docking windows within the graphic user interface
- **networks** - directory containing sample component networks
- **images** - directory containing sample images

Removing any of the mentioned files (with the exception of samples) will result in a program crash or undefined behavior.

During installation, file association will be created that ties **.ivpf** files with the framework.

5.3 Uninstallation

To uninstall the IVP Framework, go to the **Windows Control Panel — Add/Remove Programs** and choose IVP Framework.

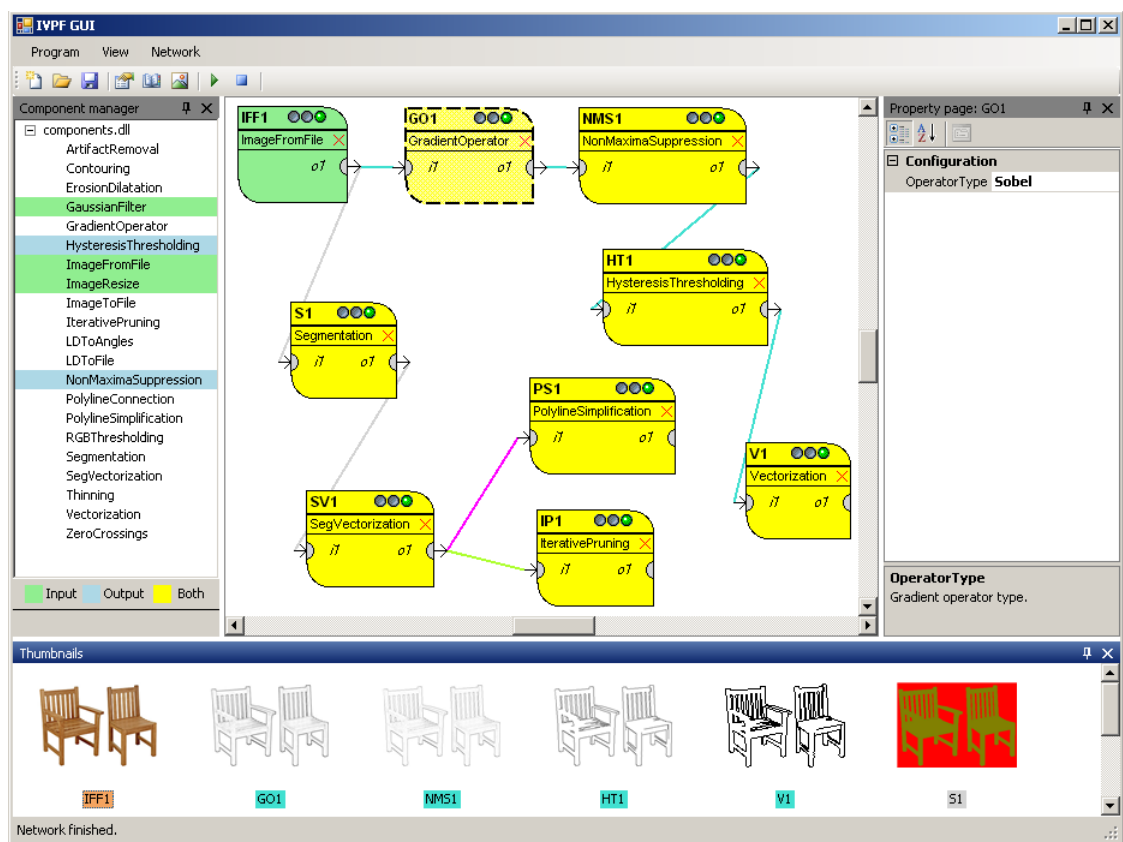


Figure 5.2: Main application window of IVP Framework’s GUI.

5.4 Program execution

5.4.1 Console Application

The console application can be run either from the command line or by double-clicking any of the framework’s document files. The exact command line syntax is `ivpf-console.exe inputFile` where `inputFile` is the name of an IVP Framework’s document.

5.4.2 Graphic User Interface Application

To run the graphic user interface, execute `ivpf-gui.exe`; main application window will appear immediately (application window is shown in Figure 5.2).

5.5 Graphic User Interface

5.5.1 Menu

The upper part of the application window contains main application menu as well as application toolbar. The functions offered by the toolbar are but a subset of those offered by the menu, with the only purpose to make their use quicker. Therefore, it suffices to give description of menu functions (see also Figure 5.3 for an image of the menu in its fully expanded state):

Program		View		Network	
New	Ctrl+N	Property Page	Ctrl+P	Run	F5
Open	Ctrl+O	Component Manager	Ctrl+M	Stop	F8
Save	Ctrl+S	Thumbnails	Ctrl+T	Run On Change	Ctrl+R
Save As...					
About					
Exit					

Figure 5.3: Expanded application menu.

Program menu contains the following items:

- **New** - creates a new empty component network; user is asked to save the current network before proceeding
- **Open** - opens an existing component network document - to choose the input path, open file dialog is shown first; user is asked to save the current network before proceeding
- **Save** - saves the current component network as an XML document - user is queried to choose output path if the network hasn't been saved yet
- **Save As** - saves the current component network - user is always queried to choose the desired output path
- **Exit** - terminates the application; user is asked to save the current network before proceeding

View menu provides user with the following functionality:

- **Component Manager** - shows the component manager panel that allows user to add components into the network as well as to read information about components
- **Property Page** - shows the property page panel which is the place for configuring component-specific parameters
- **Thumbnails** - shows the thumbnails panel that allows user to view and save generated thumbnails

Network menu contains these items:

- **Run** - runs the component network; during the execution, almost all user interface functionality is suppressed to not intervene in the execution process
- **Stop** - stops the component network; the working component (if any) is left to finish its work first
- **Run On Change** - decides whether to run network every time user makes changes to the network's structure or component parameters

5.5.2 Designer

The majority of the application's window is filled by the network designer, which is a canvas to draw components and their connections. Components are drawn as boxes, each box containing some elementary information which includes component's name, component's type as well as component's state (represented as a semaphore). Component's input and output ports are drawn as semicircles on the left (for input ports) or right side (for output ports) of the box.

Each component can be removed from the network by clicking the red cross in the upper right part of the component's box. Left double-click on an arbitrary spot of the component's box (with the exception of component's ports) will show a dialog that allows user to change the component's name. Within the user interface, component names are restricted to alfa-numeric strings starting with an alphabet letter (the framework allows to identify components with arbitrarily complex string identifiers in XML documents, this feature is however denied in the user interface to ensure smooth identifier display; if you want to use complex string identifiers, you must modify XML documents directly).

Components can be moved across the designer when clicking left mouse button and holding it. A simple left-click will merely select the component.

To begin port connection, user should left-click a component's port. Compatible ports will be highlighted immediately; to finish the port connection, user can left click any of the compatible ports. If its possible to connect the ports, it will be done; error message will be displayed otherwise. User has the chance to cancel port connection by right-clicking any spot within the empty designer space. Port disconnection is handled in the same way, i.e. by clicking the port with a right mouse button.

To view component's thumbnail in the thumbnails panel, user can select the component by double-clicking it with the right mouse button.

5.5.3 Run On Change

Running the whole network at once is often unnecessary or completely undesired. By selecting the **Run On Change** menu option, the network is run automatically from the place of last change; this happens either when changing component's properties or when connecting ports. When performing a network update, exception reporting is suppressed.

5.5.4 Component Manager Panel

Component manager panel provides basic component-related functionality. Tree of all loaded components is displayed, with top-level nodes representing loaded libraries and child nodes loaded components. Once the mouse enters component list area, input, output and worker components are highlighted unless a component is selected in the designer, in which case, component manager displays compatible components. Upon clicking component's name, component review is displayed, including short component description, port names and types.

User can add components into the component network two ways. Left-clicking the component in the manager and dragging it into the designer's area will instantiate the component, generate its new unique name, add it into the network and place its visual

representation in the designer at the position of drag drop. Second way to instantiate components is suited to create sequential networks. By double-clicking component's name in the manager, component is automatically added into the network and placed next to the last added component in the designer. If its possible to connect this newly instantiated component with the previously added one, this connection is performed automatically too.

5.5.5 Property Page Panel

Property page displays properties of the currently selected component. Property editation is test-based; first, you must select the desired property field by clicking it, which will also display short property description at the bottom of the property page. Then you can edit the respective value; finally, you must confirm it by pressing Enter key or by leaving the property page (confirmation is done automatically in some special cases such as the property page losing focus, ...). Upon entering an invalid value, dialog box is shown that informs user about the error; it's not possible to leave the property until a correct value is given.

5.5.6 Thumbnails Panel

Thumbnails panel can show outputs of some of the components. To view component's thumbnail, you must select it first into the panel by right double-clicking the corresponding component's box in the designer. When doing it, new thumbnail will appear in the panel, which will be continuously updated in harmony with the component network's operations. Double-clicking the thumbnail will open a full view window. For both the thumbnail or its opened full view, right-clicking the area will open a context menu which allows to save the image to a file. What you see is not always what gets saved into a file however: this holds only when saving thumbnails or full views of line drawings. Saving a full view of bitmap data will always save the original bitmap, no matter how it is stretched in the view window. Thumbnails can be moved across the panel while holding left mouse button. To remove a thumbnail, select it and hit the **Delete** key.

6 Comparison With Similar Programs

There is plenty of image processing and vectorization applications, varying in the level of their user-friendliness, features or availability. In this section, distinct application examples will be given along with a straightforward comparison with the IVP Framework.

WinTopo[3] and Vextractor[2] represent professional commercial products, both user-friendly and powerful. JaGrLib[1] doesn't fit into the category of vector processing applications, still it's an extensive framework based on similar ideas as the IVP Framework; JaGrLib is also a representative of an open source system from the academic sphere. It would be interesting to compare IVP Framework with the existing fully-automatical specialized systems; those are regrettably mostly unavailable, no matter if they are academic or commercial.

6.1 WinTopo 3.2 Professional

WinTopo is a professional vectorization tool. One thing that immediately catches one's eye is its maximal simplicity and user-friendliness. Almost all application window space is reserved for the view of processed image, on which user can apply variety of both image and vector processing tools.

In image processing, WinTopo beats IVP Framework almost completely, aside from ordinary image thresholding and resizing it offers the following features:

- Thresholding in HSV/HSL color space
- Brightness, contrast and gamma adjustment
- Despeckle, fill holes and prune filters
- Heal and erode filters

Edge detection is handled by multiple routines; WinTopo contains Canny edge detection method (implementation seems to be similar to that in IVP Framework) as well as "Advanced Edge Detection" option which finds edges of solid features in the image.

Thinning methods include Stentiford, Zhang-Suen and stair-removal methods and a method described as their "best combination"; to improve thinning outputs, acute angle emphasis preprocessing is available.

WinTopo employs a two-stage vectorization process. First, the raster image is thinned to one pixel lines, then the vectors are extracted. The resulting vectors are either polylines or arcs where needed/suitable (user can however suppress arc detection and rely merely on polyline representation). For polylines, WinTopo vectorization engine defaultly produces their lossless representation. Smoothing option is available to reduce sharp turns in polylines, which adjusts the vector vertices to make lines run smoothly from point to point while retaining maximum accuracy. Polyline reduction option is present to reduce the quantity of produced vectors at the cost of lesser accuracy; user can specify allowed tolerance (in tenths of pixel widths) which represents the maximum permitted deviation from the lossless vector.

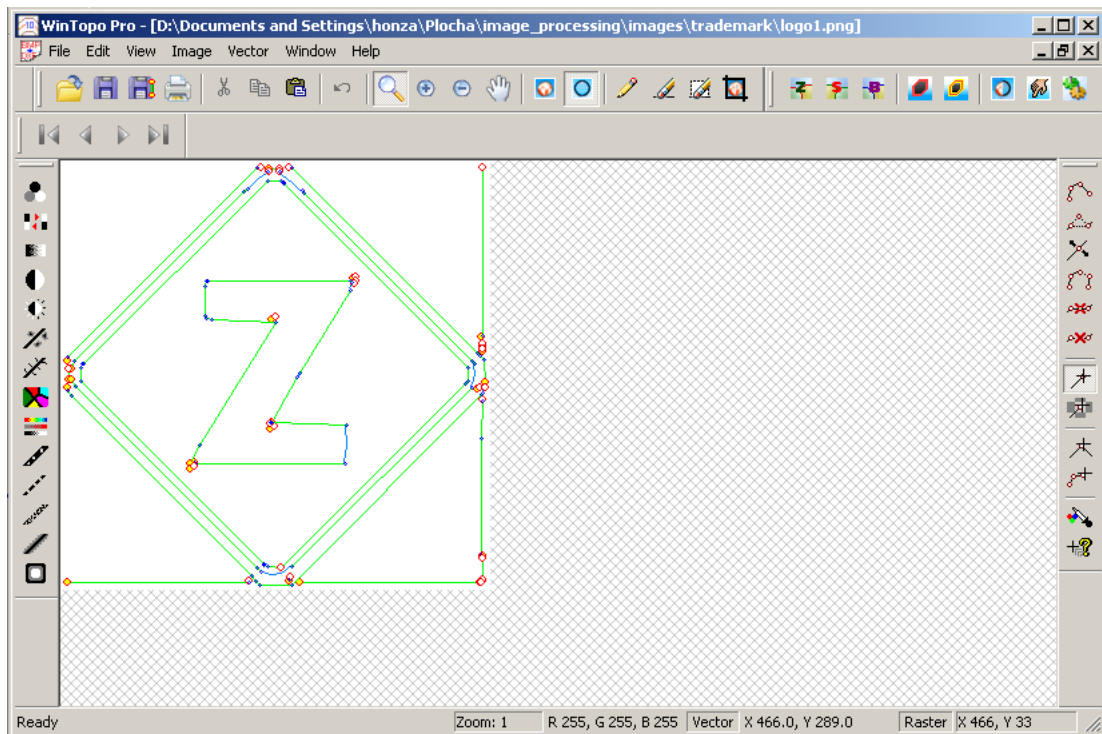


Figure 6.2: WinTopo application.

Obtained polylines can be set to break at intersections (i.e. nodes are added at intersections); the other way is to let WinTopo intelligently decide polyline's continuation through intersections. Last of all, each vector can be assigned its color.

Compared to IVP Framework, WinTopo offers no means of automatic vector data processing (such as artifact removal, polyline connection, ...), this is however more than compensated by the possibility to edit vector output directly. Allowed user operations include vector addition and removal, adding and removing vertices, vector merging, etc.

WinTopo also offers batch-processing functionality: you can specify a sequence of operations to execute on a set of input data.

As for the input and output, WinTopo supports all traditional image formats as well as a regimen of vector formats (WMF, EMF, DXF, plaintext, ...).

Screenshot of the application can be seen in Figure 6.2.

6.2 Vextractor 3.80

Vextractor is yet another example of a stand-alone vectorization application. Processed image is viewed full-size; two separate parts of the application window are kept for image's thumbnail and zoomed detail. To detect important features, Canny edge detector is implemented; alternatively, there's an opportunity to use built-in image segmentation algorithm (compared to the segmentation approach in the IVP Framework, instead of keeping only the information about segments, Vextractor associates each segment with its averaging color).

Vectorization turns the input into a set of polylines and arcs (or circles) by

tracing either center-lines or contours of input objects. The curvature of polylines is reduced to make them look smoother based on a given tolerance threshold, this approach being possibly combined with spline approximation.

Vextractor contains some handy vector processing functions. Bridges are removed to form clean junctions, dead-end polyline artifacts are removed based on their length. Disconnected contours are merged when close enough. Orthogonal line recognition mode can be enabled to correct certain shapes. The whole vectorization process is rather different from the path taken in the IVP Framework. IVP Framework doesn't bother with the quality of vectorization output, it relies on successive components - artifact handling, polyline reconnection etc. On the contrary, Vextractor works with the data more closely during the vectorization step, which subsequently results in smoother and finely polished vector result.

Vectorization scenarios can be saved and loaded separately, there is also present a nice generic set of ready-to-use scenarios (including scenarios for drawings, line arts, maps, outlines). A vectorization wizard is implemented to guide user through the vectorization process.

Obtained vector result can be fully modified at the user level. Batch-processing is incorporated within the application, it however only allows to select a single preprocessing step and a vectorization scenario; it's impossible to specify particular shape extraction steps individually.

6.3 JaGrLib

JaGrLib is a 2D/3D graphics library³ aimed for experimental and educational purposes. It seems to currently contain almost no algorithms suitable for shape extraction, hence we will focus mostly on the library's structure and its respective graphic user interface, which are both very interesting.

The overall approach taken in JaGrLib seems to be very similar to that in IVP Framework: algorithms are encapsulated within components, which can be connected to form complicated compositions. Objects are abstracted using interfaces (e.g. there are interfaces for line and arc renderers, text renderers, ...). In the IVP Framework, each component prepares its data which are read consecutively by successor components. JaGrLib mostly employs a different approach: during component execution, its data are passed immediately to the successor components.

One example to tell it all: consider a component that is supposed to draw lines. In the IVP Framework, the component would produce the desired output, store it and stop its execution. JaGrLib's component asks for an output `Bitmask` interface first, then it immediately writes its output through the interface and passes it further in the network. IVP Framework's component executes its algorithm in one defined function, JaGrLib's components are not limited to have only one place of execution; instead, component's functionality is abstracted using interfaces too (continuing our example, the component implements `LineRender` interface). This way, JaGrLib is able to employ the concept of workers - those are components that generate or produce networks's inputs (e.g. `LineWorker` randomly generates desired number of lines by writing them into its output `LineRender` port).

³JaGrLib is implemented in Java

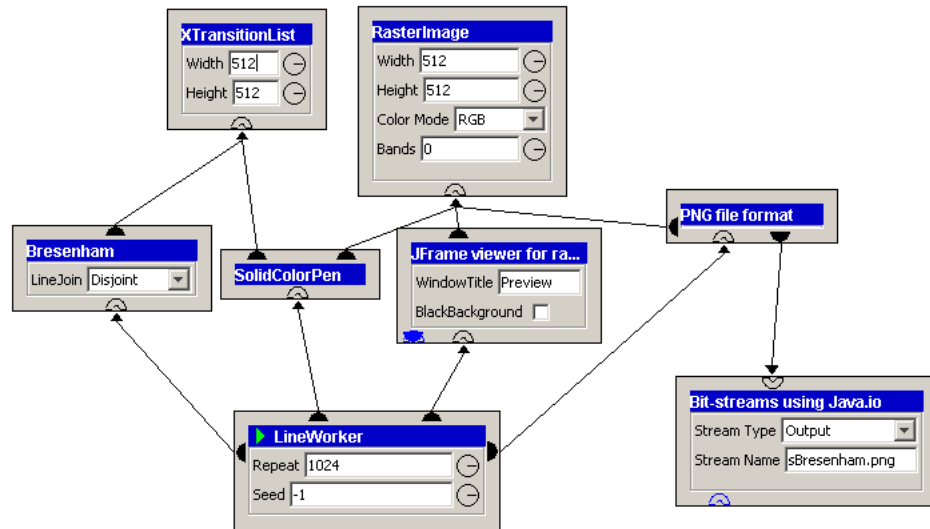


Figure 6.3: An example of JaGrLib's composition.

JaGrLib has also support for some kind of output evaluation (as already said, it's primarily aimed for educational purposes; evaluation is handy when comparing multiple versions of the same algorithm). Both the port and property registration system is purely dynamic in JaGrLib, being handled by a set of virtual functions and predefined constants.

Serialization and deserialization of components uses XML, with the format being considerably more complicated and probably even more powerful (for example, component types are stored in the document along with their parent namespaces to distinguish between components of equal names, such feature is currently unnecessary in IVP Framework considering its extent).

Compared to IVP Framework's graphic user interface, JaGrLib is rather advanced: It's possible to edit multiple compositions simultaneously, enjoying both undo and redo operations. Module registration is contained in a separate window, allowing to register or unregister components one by one. An option to view registered modules approximately equals the IVP GUI's component manager. In the design area, components are also represented as boxes (see Figure 6.3); compared to IVP Framework, component-specific parameters are contained directly within the component's box; it's possible to undock the property window and move it to a different location though. Component's ports are implemented in a clever way - their position is in no way fixed but can be changed arbitrarily which finally results in nicer and better arranged compositions. To simplify the composition even further, components can be grouped into bigger entities. For each component, user is automatically offered a list of similar components, any of the listed components being a suitable replacement for the original one.

7 Conclusion & Future Work

In the thesis, basic overview of shape extraction problems was given. Image and Vector Processing Framework was proposed as a mean to design, run and polish shape extraction methods. There is still huge amount of work ahead to achieve all the outlined goals, both framework and GUI related.

7.1 Framework-Related Improvements

What IVP Framework currently lacks is a powerful property system that would allow to exactly specify components' parameter types (such as ranges and intervals) in an easy and transparent way (e.g. using .NET metadata). Current implementation is temporary at best as it simply doesn't offer the desired level of friendliness both for component developers and potential end-users.

Structure of input and output data needs to be examined more closely in order to establish an approach to process multiple data layers; even the whole concept of interfaces to represent objects fails sometimes and as long as we want to avoid components that only convert data from one representation to another, we have to think of some trickier approach.

The major framework potential lies in an idea to extend components not only to have algorithmic functionality. At present time, the component has no means to alter what happens within the whole network. Obviously, rigid component network structure combined with fixed component parametrization can never lead to robust and widely usable shape extraction method; instead, we want the component network to evaluate outputs simultaneously. By implementing specialized components that measure quality of produced outputs and set the network accordingly, one could possibly hope to get good results even on large input data sets. Similarly, components are presently overmuch limited to contain fixed algorithmic functionality, it could be advantageous to use some scripting language to directly specify what happens within the component.

7.2 GUI-Related Improvements

Some of the possible GUI improvements are listed below:

- Improved set of output editing tools (especially for line drawings, manual output refinement often comes handy)
- Built-in batch processing (currently, batch processing is naturally available via input XML document edition, i.e. by using some standalone macro-processing tools)

References

- [1] Jagrlib, <http://www.cgg.ms.mff.cuni.cz/jagrlib>.
- [2] Vextractor, <http://www.vextrasoft.com>.
- [3] Wintopo, <http://wintopo.com>.
- [4] S. V. Ablameyko. *Introduction to Interpretation of Graphic Images*. SPIE, 1997.
- [5] P. Altman. Digitalization of map. Master's thesis, Charles University, Department of Software and Computer Science Education, 2004.
- [6] J. Canny. A computational approach to edge detection. *IEEE Trans. Pattern Anal. Mach. Intell.*, 8(6):679–698, 1986.
- [7] D. H. Douglas and T. K. Peucker. Algorithms for the reduction of the number of points required to represent a line or its caricature. *Canadian Cartographer*, 10(2):112–122, 1973.
- [8] J. Klíma and T. Skopal. Shape extraction framework for similarity search in image databases. In *Proceedings of the DATESO 2007 Annual International Workshop on Databases, TExts, Specifications and Objects*, pages 89–102, 2007.
- [9] P. L. Rosin and G. A. W. West. Segmentation of edges into lines and arcs. *Image Vision Comput.*, 7(2):109–114, 1989.
- [10] J. Sklansky and V. Gonzalez. Fast polygonal approximation of digitized curves. *Pattern Recognition*, 12(5):327–331, 1980.
- [11] F. Stentiford and R. Mortimer. Some new heuristics for thinning binary hand-printed characters for ocr. *IEEE Transactions on Systems Man and Cybernetics*, 13(1):81–84, 1983.
- [12] K. Wall and P.-E. Danielsson. A fast sequential method for polygonal approximation of digitized curves. *Comput. Vision Graph. Image Process.*, 28(3):220–227, 1984.
- [13] T. Y. Zhang and C. Y. Suen. A fast parallel algorithm for thinning digital patterns. *Commun. ACM*, 27(3):236–239, 1984.

A XML document example

```
<xml>
  <components>
    <component>
      <name>IFF1</name>
      <type>ImageFromFile</type>
      <parameter>
        <name>Path</name>
        <value>lena . jpg</value>
      </parameter>
    </component>
    <component>
      <name>S1</name>
      <type>Segmentation</type>
      <parameter>
        <name>NumberOfSegments</name>
        <value>3</value>
      </parameter>
      <parameter>
        <name>SegmentationType</name>
        <value>KMeansRGB</value>
      </parameter>
    </component>
    <component>
      <name>SV1</name>
      <type>SegVectorization</type>
    </component>
    <component>
      <name>ITF1</name>
      <type>ImageToFile</type>
      <parameter>
        <name>Path</name>
        <value>output . dxf</value>
      </parameter>
    </component>
  </components>
  <connections>
    <connection>
      <input_component>S1</input_component>
      <output_component>IFF1</output_component>
      <input_port>i1</input_port>
      <output_port>o1</output_port>
    </connection>
    <connection>
      <input_component>SV1</input_component>
      <output_component>S1</output_component>
      <input_port>i1</input_port>
      <output_port>o1</output_port>
    </connection>
  </connections>
  <gui>
    <designer>
      <hminimum>-2000</hminimum>
      <hmaximum>2000</hmaximum>
      <vminimum>-2000</vminimum>
```

```

    <vmaximum>2000</vmaximum>
    <posx>-301</posx>
    <posy>-242</posy>
    <component_panel>
      <name>IFF1</name>
      <top>-232</top>
      <left>-291</left>
    </component_panel>
    <component_panel>
      <name>S1</name>
      <top>-232</top>
      <left>-159</left>
    </component_panel>
    <component_panel>
      <name>SV1</name>
      <top>-232</top>
      <left>-30</left>
    </component_panel>
    <component_panel>
      <name>ITF1</name>
      <top>-232</top>
      <left>117</left>
    </component_panel>
  </designer>
  <thumbnails>
    <thumbnail>IFF1</thumbnail>
    <thumbnail>S1</thumbnail>
    <thumbnail>SV1</thumbnail>
  </thumbnails>
</gui>
</xml>

```